Java

Introduction to Java

Java is a high-level, object-oriented programming language developed by **James Gosling** at Sun Microsystems (now owned by Oracle) in 1995. It is designed to be **platform-independent** using the concept of "Write Once, Run Anywhere" (WORA). This is achieved through the **Java Virtual Machine (JVM)**, which allows Java programs to run on any operating system.

Key Features of Java:

1. **Platform Independence:** Java code is compiled into **bytecode**, which runs on any device with a JVM.
2. **Object-Oriented:** Java follows the object-oriented programming (OOP) paradigm.
3. **Robust and Secure:** It has strong memory management and security features.
4. **Multithreading:** Supports multiple threads of execution.
5. **Garbage Collection:** Automatic memory management via the **Garbage Collector**.
6. **Rich API & Libraries:** Java comes with extensive libraries for networking, file handling, and more.

Overview of Java Platform and Editions

Java has multiple editions tailored for different application areas:

1. **Java Standard Edition (Java SE):**
    - Core Java libraries and APIs.
    - Used for developing desktop applications and foundational Java programs.
    - Includes the **JDK (Java Development Kit)** and **JRE (Java Runtime Environment)**.
2. **Java Enterprise Edition (Java EE) (Now Jakarta EE):**
    - Used for developing web-based, enterprise-level applications.
    - Includes additional APIs like Servlets, JSP, JPA, and EJB.
3. **Java Micro Edition (Java ME):**

- Used for developing mobile and embedded systems.
- Lightweight version of Java SE with specialized APIs.

4. **JavaFX:**
    - Used for developing rich internet applications with advanced graphical interfaces.

Installation and Setup of Java Development Environment (JDK 17)

Steps to Install JDK 17:

1. **Download JDK 17:**
    - Visit Oracle's official site or OpenJDK.
2. **Install JDK:**
    - Follow the installation instructions for your OS (Windows, macOS, or Linux).
    - Set the **JAVA_HOME** environment variable.
3. **Verify Installation:**
    - Open a terminal or command prompt.
    - Run:
    - java -version
    -
    - You should see something like:
    - java version "17.0.1" 2021-10-19 LTS
    -

Introduction to Java Development Kit (JDK), Java Runtime Environment (JRE), and JVM

Java Development Kit (JDK)

- A **software development kit (SDK)** that contains tools for **writing, compiling, and debugging** Java programs.
- Includes:

    - **JRE (Java Runtime Environment)**
    - **Compiler (javac)**
    - **Java Debugger (jdb)**
    - **Other development tools**

Java Runtime Environment (JRE)

- A subset of the JDK that includes only the tools needed to **run** Java applications (not develop them).
- Includes:
    - **JVM (Java Virtual Machine)**
    - **Core Java Libraries**
    - **Runtime Libraries**

Java Virtual Machine (JVM)

- A key component of Java that interprets and executes Java bytecode.
- Converts bytecode into **machine code** for the specific platform.
- Provides features like:
    - **Garbage Collection**
    - **Just-In-Time (JIT) Compilation**
    - **Security Management**

Writing, Compiling, and Running Java Programs

1. **Write Java Code (HelloWorld.java)**
2. public class HelloWorld {
3. public static void main(String[] args) {
4. System.out.println("Hello, World!");
5. }
6. }
7.
8. **Compile the Program**

- Run:
- javac HelloWorld.java
-
- Generates HelloWorld.class (bytecode).

9. **Run the Program**
    - Execute:
    - java HelloWorld
    -
    - Output:
    - Hello, World!
    -

Basics of Java Programming

Data Types and Variables

Primitive Data Types in Java

Data Type	Size	Default Value	Description
byte	1 byte	0	Stores integers (-128 to 127)
short	2 bytes	0	Stores integers (-32,768 to 32,767)
int	4 bytes	0	Stores integers (-2^31 to 2^31-1)
long	8 bytes	0L	Stores large integers (-2^63 to 2^63-1)
float	4 bytes	0.0f	Stores decimal numbers (single precision)
double	8 bytes	0.0d	Stores decimal numbers (double precision)
char	2 bytes	'\u0000'	Stores a single character
boolean	1 bit	false	Stores true or false

Variables in Java

```
int age = 25;
double price = 99.99;
```

```
boolean isJavaFun = true;

char grade = 'A';
```

Operators in Java

Types of Operators

1. **Arithmetic Operators:** +, , , /, %
2. **Relational Operators:** ==, !=, >, <, >=, <=
3. **Logical Operators:** &&, ||, !
4. **Bitwise Operators:** &, |, ^, ~, <<, >>
5. **Assignment Operators:** =, +=, =, =, /=
6. **Ternary Operator:** condition ? value1 : value2

Control Statements

1. **Conditional Statements**
2. if (age > 18) {
3. System.out.println("Adult");
4. } else {
5. System.out.println("Minor");
6. }
7.
8. **Loops**
9. for (int i = 0; i < 5; i++) {
10. System.out.println("Iteration: " + i);
11. }
12.

Object-Oriented Programming (OOP) in Java

1. Classes and Objects

```
class Car {
   String brand;
   int speed;

   void display() {
      System.out.println(brand + " runs at " + speed + " km/hr");
   }
}

public class Main {
   public static void main(String[] args) {
      Car myCar = new Car();
      myCar.brand = "Tesla";
      myCar.speed = 200;
      myCar.display();
   }
}
```

2. Encapsulation

- Protects data using **private** access specifiers.

```
class BankAccount {
   private double balance;

   public void setBalance(double amount) {
      balance = amount;
   }
```

```
    public double getBalance() {
        return balance;
    }
}
```

3. Inheritance

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

4. Polymorphism

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow");
```

```
    }
}
```

5. Abstraction and Interfaces

- **Abstract Class**

```
abstract class Vehicle {
    abstract void start();
}
```

- **Interface**

```
interface Animal {
    void sound();
}
```

This covers the foundational Java concepts in depth! Let me know if you need any further explanations. 🚀

Exception Handling in Java

Introduction to Exceptions

An **exception** in Java is an event that occurs during program execution and disrupts the normal flow of instructions. Exceptions can be caused by:

- **User Input Errors**
- **Hardware Failures**
- **Network Issues**
- **Logical Errors in the Code**

Java provides a robust mechanism to handle these exceptions using the **Exception Handling** framework.

Types of Exceptions

1. **Checked Exceptions** (Compile-time)
    - Must be handled using try-catch or declared using throws.

- Examples:
  - IOException (File not found, etc.)
  - SQLException (Database errors)

2. **Unchecked Exceptions** (Runtime)
   - Occur due to logical mistakes and need proper coding practices to avoid them.
   - Examples:
     - NullPointerException
     - ArrayIndexOutOfBoundsException
     - ArithmeticException
3. **Errors** (Critical System Issues)
   - These are not meant to be handled in normal scenarios.
   - Examples:
     - OutOfMemoryError
     - StackOverflowError

Handling Exceptions

Java provides try, catch, finally, throw, and throws for exception handling.

1. Try-Catch Block

```
public class ExceptionExample {
  public static void main(String[] args) {
    try {
      int result = 10 / 0; // ArithmeticException
      System.out.println(result);
    } catch (ArithmeticException e) {
      System.out.println("Cannot divide by zero.");
    }
  }
```

```
}
```

2. Multiple Catch Blocks

```
try {
   int[] numbers = {1, 2, 3};
   System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException
} catch (ArithmeticException e) {
   System.out.println("Arithmetic Error");
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Array index is out of bounds!");
}
```

3. Finally Block (Always Executes)

```
try {
   int num = Integer.parseInt("ABC"); // NumberFormatException
} catch (NumberFormatException e) {
   System.out.println("Invalid number format!");
} finally {
   System.out.println("This block always executes.");
}
```

4. Throwing Exceptions (throw keyword)

```
class Test {
   static void checkAge(int age) {
      if (age < 18) {
         throw new ArithmeticException("Not eligible to vote");
      } else {
         System.out.println("Eligible to vote");
```

```
    }
  }

  public static void main(String[] args) {
    checkAge(16); // Throws an exception
  }
}
```

5. Declaring Exceptions (throws keyword)

```
class FileExample {
  void readFile() throws IOException {
    FileReader file = new FileReader("test.txt");
  }
}
```

Custom Exceptions

Java allows us to create our own **custom exceptions** by extending the Exception class.

```
class CustomException extends Exception {
  public CustomException(String message) {
    super(message);
  }
}

public class Test {
  public static void main(String[] args) {
    try {
      throw new CustomException("This is a custom exception.");
```

```
    } catch (CustomException e) {
        System.out.println(e.getMessage());
    }
  }
}
```

Java Collections Framework (JCF)

The **Java Collections Framework (JCF)** provides a set of data structures and interfaces to store and manipulate groups of objects efficiently.

Hierarchy of Java Collections

1. **Collection Interface**
    - **List** (Ordered, Duplicates Allowed)
    - **Set** (Unique Elements, Unordered)
    - **Queue** (FIFO Ordering)
2. **Map Interface** (Key-Value Pairs)

List Interface and Implementations

Lists maintain insertion order and allow duplicates.

1. ArrayList (Resizable Array)

- Dynamic array implementation.
- Fast for retrieving elements but slow for inserting/deleting in the middle.

```
import java.util.*;

public class ListExample {
  public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("Apple");
```

```
        list.add("Banana");
        list.add("Mango");

        System.out.println(list); // Output: [Apple, Banana, Mango]
    }
}
```

2. LinkedList (Doubly Linked List)

- Fast insertion and deletion.
- Slow random access.

```
List<String> linkedList = new LinkedList<>();
linkedList.add("Dog");
linkedList.add("Cat");
System.out.println(linkedList);
```

3. Vector (Thread-Safe List)

- Synchronized version of ArrayList.

```
List<Integer> vector = new Vector<>();
vector.add(10);
vector.add(20);
System.out.println(vector);
```

Set Interface and Implementations

Sets store **unique elements** (no duplicates).

1. HashSet (Unordered, Unique Elements)

```
Set<Integer> hashSet = new HashSet<>();
hashSet.add(1);
```

```
hashSet.add(2);
hashSet.add(2); // Duplicate ignored
System.out.println(hashSet);
```

2. TreeSet (Sorted Order)

```
Set<Integer> treeSet = new TreeSet<>();
treeSet.add(5);
treeSet.add(3);
treeSet.add(10);
System.out.println(treeSet); // Output: [3, 5, 10]
```

3. LinkedHashSet (Maintains Insertion Order)

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Java");
linkedHashSet.add("Python");
linkedHashSet.add("C++");
System.out.println(linkedHashSet);
```

Map Interface and Implementations

Maps store **key-value pairs**.

1. HashMap (Unordered Key-Value Pairs)

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("A", 1);
hashMap.put("B", 2);
System.out.println(hashMap);
```

2. TreeMap (Sorted Keys)

```
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("Banana", 2);
treeMap.put("Apple", 1);
System.out.println(treeMap);
```

3. LinkedHashMap (Maintains Insertion Order)

```
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("Zebra", 3);
linkedHashMap.put("Tiger", 2);
System.out.println(linkedHashMap);
```

Queue Interface and Implementations

Queues follow **FIFO (First-In-First-Out)**.

1. PriorityQueue (Heap Implementation)

```
Queue<Integer> queue = new PriorityQueue<>();
queue.add(5);
queue.add(1);
queue.add(10);
System.out.println(queue.poll()); // Smallest element first
```

2. Deque (Double-Ended Queue)

```
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("First");
deque.addLast("Last");
System.out.println(deque);
```

Stream API (Java 8+)

Streams allow processing collections in a **functional programming style**.

1. Filtering Elements

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

2. Mapping Elements

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.stream().map(String::toUpperCase).forEach(System.out::println);
```

3. Sorting

```
List<Integer> list = Arrays.asList(5, 2, 8, 1);

list.stream().sorted().forEach(System.out::println);
```

Conclusion

- **Exception Handling** ensures programs can recover from errors.
- **Collections Framework** provides efficient data structures.
- **Stream API** simplifies collection processing.

Would you like code examples on any specific topic? 🚀

Functional Programming in Java

Functional programming is a paradigm where computations are treated as **evaluations of mathematical functions** without changing state or mutable data. Java introduced **functional programming features** in **Java 8** to enhance code readability and efficiency.

Key Features of Functional Programming in Java

1. **Lambda Expressions** - Anonymous functions to simplify code.
2. **Functional Interfaces** - Interfaces with a single abstract method, used in lambda expressions.
3. **Method References** - Shorter syntax for calling existing methods.

4. **Optional Class -** Helps to handle null values effectively.
5. **Streams API** - Enables functional-style operations on collections.
6. **Parallel Streams -** Used to process data concurrently for better performance.

---

1. Lambda Expressions

A **lambda expression** is a short way to define a method that can be passed as an argument.

Syntax of Lambda Expression

(parameters) -> { body }

Example: Using Lambda in Java

```
// Without Lambda
interface Greeting {
   void sayHello();
}

public class LambdaExample {
   public static void main(String[] args) {
      Greeting greeting = new Greeting() {
         public void sayHello() {
            System.out.println("Hello, World!");
         }
      };
      greeting.sayHello();
   }
}

// With Lambda
```

```
interface Greeting {
    void sayHello();
}

public class LambdaExample {
    public static void main(String[] args) {
        Greeting greeting = () -> System.out.println("Hello, World!");
        greeting.sayHello();
    }
}
```

Lambda with Parameters

```
interface MathOperation {
    int operate(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        MathOperation sum = (a, b) -> a + b;
        System.out.println(sum.operate(5, 10)); // Output: 15
    }
}
```

---

2. Functional Interfaces

A **functional interface** is an interface with only **one abstract method**, making it ideal for lambda expressions.

Examples of Built-in Functional Interfaces

1. Predicate<T> - Returns true or false based on a condition.
2. Function<T, R> - Takes an input and returns a transformed output.
3. Consumer<T> - Takes an input but **does not return anything**.
4. Supplier<T> - Does **not take input** but returns a value.

Example: Predicate Interface

```
import java.util.function.Predicate;

public class PredicateExample {
   public static void main(String[] args) {
      Predicate<Integer> isEven = num -> num % 2 == 0;
      System.out.println(isEven.test(10)); // Output: true
   }
}
```

Example: Function Interface

```
import java.util.function.Function;

public class FunctionExample {
   public static void main(String[] args) {
      Function<String, Integer> lengthFunction = str -> str.length();
      System.out.println(lengthFunction.apply("Lambda")); // Output: 6
   }
}
```

Example: Consumer Interface

```
import java.util.function.Consumer;

public class ConsumerExample {
```

```
    public static void main(String[] args) {
        Consumer<String> printMessage = message -> System.out.println(message);
        printMessage.accept("Hello, Functional Programming!");
    }
}
```

Example: Supplier Interface

```
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomNumber = () -> Math.random();
        System.out.println(randomNumber.get()); // Output: Random number
    }
}
```

---

3. Method References

Method references are a shorthand notation for calling **existing methods** using :: operator.

Types of Method References

1. **Static Method Reference:** ClassName::staticMethod
2. **Instance Method Reference:** object::instanceMethod
3. **Constructor Reference:** ClassName::new

Example: Static Method Reference

```
import java.util.function.Consumer;

public class MethodRefExample {
```

```
    public static void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        Consumer<String> printer = MethodRefExample::printMessage;
        printer.accept("Hello, Method Reference!");
    }
}
```

Example: Instance Method Reference

```
class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }
}

public class MethodRefExample {
    public static void main(String[] args) {
        Printer printer = new Printer();
        Consumer<String> printMessage = printer::print;
        printMessage.accept("Instance Method Reference");
    }
}
```

---

4. Optional Class

The Optional<T> class helps handle null values gracefully, avoiding NullPointerException.

Example: Handling Null with Optional

```
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        Optional<String> optional = Optional.ofNullable(null);

        // Using orElse to provide a default value
        System.out.println(optional.orElse("Default Value"));
    }
}
```

Example: Using ifPresent()

```
Optional<String> optional = Optional.of("Hello");
optional.ifPresent(System.out::println); // Output: Hello
```

---

5. Streams API

Streams allow **functional-style operations** on collections like **filter, map, reduce**.

Example: Filtering Even Numbers

```
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
        numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
    }
}
```

Example: Mapping to Uppercase

```
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.stream().map(String::toUpperCase).forEach(System.out::println);
    }
}
```

Example: Summing a List of Numbers

```
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = numbers.stream().reduce(0, Integer::sum);
        System.out.println(sum); // Output: 15
    }
}
```

---

6. Parallel Streams

Parallel streams **divide tasks** into multiple CPU cores, improving performance.

Example: Parallel Processing

```
import java.util.Arrays;
import java.util.List;


public class ParallelStreamExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);


      numbers.parallelStream().forEach(System.out::println);
   }
}
```

Conclusion

- **Lambda Expressions** reduce boilerplate code.
- **Functional Interfaces** allow flexible programming.
- **Method References** provide a concise way to call existing methods.
- **Optional Class** prevents NullPointerException.
- **Streams API** provides powerful data processing.
- **Parallel Streams** improve performance by utilizing multiple cores.

Would you like any additional explanations or hands-on exercises? 🚀

Java I/O and File Handling

Java provides robust support for **input and output (I/O) operations**, allowing programs to read from and write to **files, networks, and streams**. Java's **I/O API** includes classes for handling various types of data streams and files efficiently.

---

1. Java I/O Streams

What is a Stream?

A **stream** in Java represents a **sequence of data**. Java's **I/O streams** can be classified into:

1. **Byte Streams -** Read and write data **byte-by-byte** (InputStream & OutputStream).
2. **Character Streams -** Read and write data **character-by-character** (Reader & Writer).

Byte Streams (For Binary Data)

Byte streams are used for handling **binary files** (images, audio, videos, etc.).

They work with **8-bit data (1 byte)**.

Example: Reading a File (Byte Stream)

```
import java.io.FileInputStream;
import java.io.IOException;


public class ByteStreamExample {
   public static void main(String[] args) {
      try (FileInputStream fis = new FileInputStream("sample.txt")) {
         int data;
         while ((data = fis.read()) != -1) {
            System.out.print((char) data); // Print character by character
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Example: Writing a File (Byte Stream)

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String content = "Hello, Java I/O!";
            fos.write(content.getBytes()); // Convert string to bytes and write
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Character Streams (For Text Data)

Character streams work with **16-bit Unicode characters**, making them suitable for **text-based** files.

Example: Reading a File (Character Stream)

```
import java.io.FileReader;
import java.io.IOException;

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("sample.txt")) {
            int data;
            while ((data = fr.read()) != -1) {
```

```
            System.out.print((char) data);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Example: Writing a File (Character Stream)

```
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamExample {
  public static void main(String[] args) {
    try (FileWriter fw = new FileWriter("output.txt")) {
      fw.write("Hello, Java File Handling!");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

---

2. File Handling with java.nio.file Package

The java.nio.file package (introduced in Java 7) provides an **efficient and modern way** to work with files.

Key Classes in java.nio.file

- **Paths -** Used to create Path objects.

- **Files -** Contains methods to perform operations on files (read, write, copy, move, delete).

Example: Reading a File using Files.readAllLines()

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
import java.util.List;

public class NIOExample {
   public static void main(String[] args) {
      try {
         List<String> lines = Files.readAllLines(Path.of("sample.txt"));
         lines.forEach(System.out::println);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Example: Writing to a File using Files.write()

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
import java.util.Arrays;

public class NIOExample {
   public static void main(String[] args) {
      try {
```

```
            Files.write(Path.of("output.txt"), Arrays.asList("Hello, Java NIO!"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example: Copying a File

```
import java.nio.file.*;

public class FileCopyExample {
    public static void main(String[] args) {
        try {
            Path source = Paths.get("source.txt");
            Path destination = Paths.get("destination.txt");
            Files.copy(source, destination, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File copied successfully!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example: Deleting a File

```
import java.nio.file.*;

public class FileDeleteExample {
    public static void main(String[] args) {
```

```
        try {
            Files.deleteIfExists(Paths.get("sample.txt"));
            System.out.println("File deleted successfully!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

---

3. Serialization and Deserialization

Serialization is the process of **converting an object into a byte stream** so it can be saved to a file or sent over a network.

Deserialization is the process of **converting a byte stream back into an object**.

Steps to Perform Serialization

1. The class **must implement** the Serializable interface.
2. Use **ObjectOutputStream** to write the object.
3. Use **ObjectInputStream** to read the object.

Example: Serialization

```
import java.io.*;

// Serializable class
class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;
```

```
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("student.ser"))) {
            Student student = new Student("Alice", 22);
            oos.writeObject(student); // Writing object to file
            System.out.println("Object Serialized!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example: Deserialization

```
import java.io.*;

public class DeserializationExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("student.ser"))) {
            Student student = (Student) ois.readObject(); // Reading object from file
            System.out.println("Deserialized Object: " + student.name + ", " + student.age);
```

```
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
  }
}
```

Summary

Feature	Description
Byte Streams	Read/write binary data (images, audio, etc.)
Character Streams	Read/write text files (UTF-16 support)
java.nio.file API	Modern file handling (read, write, copy, delete)
Serialization	Convert objects to byte streams for storage or transfer
Deserialization	Convert byte streams back to objects

Would you like additional explanations or practice exercises? 🚀

Multithreading and Concurrency in Java

Multithreading in Java allows concurrent execution of multiple tasks (threads), making programs **faster, efficient, and responsive**. Java provides built-in support for multithreading through the Thread class and the Runnable interface.

---

1. Introduction to Multithreading

What is Multithreading?

Multithreading is a programming technique where multiple **threads** execute **simultaneously** within a single process.

- A **thread** is the smallest unit of execution.
- Java supports **preemptive multitasking**, where the OS scheduler decides which thread runs.
- Threads in Java run **independently**, allowing parallel execution.

Advantages of Multithreading

✅ Better CPU Utilization

✅ Faster execution (Parallelism)

✅ Improved performance in multi-core systems

✅ Enhanced responsiveness in applications (e.g., GUI, web servers)

---

2. Thread Lifecycle and Thread Control

Each thread goes through different **stages** in its lifecycle:

Thread Lifecycle in Java

1. **New (Created) -** The thread is created using Thread class but not yet started.
2. **Runnable -** The thread is ready to run and waiting for CPU time.
3. **Running -** The thread is executing its task.
4. **Blocked/Waiting -** The thread is **paused** due to some condition (e.g., waiting for a resource).
5. **Terminated (Dead) -** The thread has finished execution.

Creating and Running Threads

Threads in Java can be created in **two ways**:

Method 1: Extending the Thread Class

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Thread is running...");
  }
}

public class ThreadExample {
  public static void main(String[] args) {
```

```
        MyThread t1 = new MyThread();
        t1.start(); // Start the thread
    }
}
```

Method 2: Implementing the Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable...");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

Best Practice:

Using Runnable is **preferred** because Java **does not support multiple inheritance**, so implementing an interface provides more flexibility.

Thread Methods for Control

Method	Description
start()	Starts the thread execution

Method	Description
run()	Contains the task to be executed
sleep(ms)	Puts the thread to sleep for ms milliseconds
join()	Waits for the thread to finish before continuing
yield()	Allows other threads to execute
interrupt()	Interrupts a thread

Example: Using join()

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println(Thread.currentThread().getName() + " - " + i);
      }
   }
}

public class ThreadJoinExample {
   public static void main(String[] args) throws InterruptedException {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();

      t1.start();
      t1.join();  // Main thread waits for t1 to finish
      t2.start();
   }
}
```

---

3. Synchronization in Java

Problem:

When multiple threads access a shared resource **simultaneously**, it may lead to **race conditions** and **inconsistent results**.

Example of Race Condition

```
class Counter {
   int count = 0;
   public void increment() { count++; }
}

public class RaceConditionExample {
   public static void main(String[] args) {
      Counter counter = new Counter();

      Thread t1 = new Thread(() -> {
         for (int i = 0; i < 1000; i++) counter.increment();
      });

      Thread t2 = new Thread(() -> {
         for (int i = 0; i < 1000; i++) counter.increment();
      });

      t1.start();
      t2.start();

      System.out.println("Final Count: " + counter.count); // May print incorrect value
   }
```

```
}
```

Solution: Synchronization

Using the synchronized keyword ensures only **one thread** accesses the method at a time.

```
class Counter {
  int count = 0;

  public synchronized void increment() { count++; }
}

public class SynchronizationExample {
  public static void main(String[] args) throws InterruptedException {
    Counter counter = new Counter();

    Thread t1 = new Thread(() -> {
      for (int i = 0; i < 1000; i++) counter.increment();
    });

    Thread t2 = new Thread(() -> {
      for (int i = 0; i < 1000; i++) counter.increment();
    });

    t1.start();
    t2.start();
    t1.join();
    t2.join();
```

```
        System.out.println("Final Count: " + counter.count); // Correct output: 2000
    }
}
```

---

4. Concurrency Utilities (java.util.concurrent Package)

Java provides advanced **concurrent utilities** for thread management:

Key Classes in java.util.concurrent

Class/Interface	Description
ExecutorService	Manages a pool of threads
Callable<T>	Similar to Runnable, but returns a result
Future<T>	Represents the result of an asynchronous task
CountDownLatch	Makes a thread wait for others to complete
Semaphore	Limits access to a resource
ConcurrentHashMap	A thread-safe HashMap

Example: Using ExecutorService for Thread Pooling

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3); // 3 threads

        for (int i = 1; i <= 5; i++) {
            final int taskId = i;
            executor.execute(() -> {
```

```
            System.out.println("Task " + taskId + " executed by " +
Thread.currentThread().getName());
        });
    }

    executor.shutdown(); // Shutdown executor
  }
}
```

Example: Using Callable and Future

Callable<T> allows tasks to return a result.

```
import java.util.concurrent.*;

public class CallableExample {
  public static void main(String[] args) throws Exception {
    ExecutorService executor = Executors.newFixedThreadPool(2);

    Callable<Integer> task = () -> {
      Thread.sleep(1000);
      return 42;
    };

    Future<Integer> result = executor.submit(task);

    System.out.println("Task submitted...");
    System.out.println("Result: " + result.get()); // Blocks until task completes
```

```
        executor.shutdown();
    }
}
```

Summary

Feature	Description
Thread Class	Create a thread by extending Thread
Runnable Interface	Preferred way to create threads
Synchronization	Prevents race conditions using synchronized
ExecutorService	Manages thread pools for efficient task execution
Callable & Future	Allows returning values from threads

Would you like additional explanations or hands-on coding exercises? 🚀

Introduction to Reactive Programming in Java

Reactive Programming is an **asynchronous programming paradigm** focused on handling **data streams and event-driven architecture**. It allows **non-blocking execution**, making applications more **responsive, scalable, and efficient**.

---

1. Reactive Programming Fundamentals

What is Reactive Programming?

Reactive programming is based on **reacting** to data changes **as they happen**, rather than manually pulling updates.

✅ **Asynchronous**: Non-blocking operations

✅ **Event-driven**: Responds to events dynamically

✅ **Streams-based**: Processes data as a continuous flow

Reactive Manifesto (Key Principles)

- **Responsive** - Quick response times
- **Resilient** - Handles failures gracefully

- **Elastic** - Scales efficiently
- **Message-Driven** - Uses event streams

---

2. Reactive Streams and Backpressure

What are Reactive Streams?

Reactive Streams define a **standard API** for asynchronous **data processing**.

It consists of **four core interfaces**:

Component	Role
Publisher<T>	Emits a stream of data items
Subscriber<T>	Consumes data from Publisher
Subscription	Manages backpressure
Processor<T,R>	Transforms data (both Publisher & Subscriber)

Understanding Backpressure

Backpressure occurs when the **data producer (Publisher)** generates items **faster** than the **consumer (Subscriber)** can process them.

Solution: The Subscriber can request **how many items** it can handle at a time.

```
import org.reactivestreams.*;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;


public class SimpleReactiveExample {
  public static void main(String[] args) {
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();


    Flow.Subscriber<String> subscriber = new Flow.Subscriber<>() {
      private Flow.Subscription subscription;
      @Override
```

```
        public void onSubscribe(Flow.Subscription subscription) {
            this.subscription = subscription;
            subscription.request(1); // Request 1 item at a time (backpressure handling)
        }
        @Override
        public void onNext(String item) {
            System.out.println("Received: " + item);
            subscription.request(1); // Request next item
        }
        @Override
        public void onError(Throwable throwable) {
            System.err.println("Error: " + throwable.getMessage());
        }
        @Override
        public void onComplete() {
            System.out.println("Completed!");
        }
    };

    publisher.subscribe(subscriber);
    publisher.submit("Hello, Reactive World!");
    publisher.close();
  }
}
```

---

3. Project Reactor Basics (Reactive Programming in Java)

Project Reactor is a Java library implementing the **Reactive Streams API**.

It provides two **main types** for handling reactive data streams:

Type	Description
Mono<T>	Emits **0 or 1** item
Flux<T>	Emits **0 to N** items (stream of data)

Example: Using Mono

```
import reactor.core.publisher.Mono;

public class MonoExample {
   public static void main(String[] args) {
      Mono<String> mono = Mono.just("Hello, Reactor!");
      mono.subscribe(System.out::println);
   }
}
```

Example: Using Flux

```
import reactor.core.publisher.Flux;

public class FluxExample {
   public static void main(String[] args) {
      Flux<String> flux = Flux.just("Apple", "Banana", "Cherry");
      flux.subscribe(System.out::println);
   }
}
```

Transforming Data with map() and flatMap()

```
import reactor.core.publisher.Flux;
```

```
public class TransformExample {
    public static void main(String[] args) {
        Flux<Integer> numbers = Flux.range(1, 5)
            .map(n -> n * 2); // Multiply each number by 2

        numbers.subscribe(System.out::println);
    }
}
```

---

4. Concurrency in Reactive Programming

Why Concurrency in Reactive Programming?

Reactive programming inherently supports **asynchronous execution**, making it **highly efficient** for handling **parallel tasks**.

Schedulers in Reactor

Schedulers define **how and where** work is executed:

Scheduler	Description
Schedulers.immediate()	Runs tasks on the current thread
Schedulers.single()	Runs tasks on a single, dedicated thread
Schedulers.parallel()	Uses a pool of parallel threads
Schedulers.boundedElastic()	Expands the thread pool as needed

Example: Running Tasks on Parallel Threads

```
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;

public class SchedulerExample {
    public static void main(String[] args) throws InterruptedException {
        Flux.range(1, 5)
```

```
        .map(i -> {
            System.out.println("Processing " + i + " on thread: " +
Thread.currentThread().getName());
            return i;
        })
        .subscribeOn(Schedulers.parallel())
        .subscribe();

    Thread.sleep(1000); // Allow time for async tasks to complete
  }
}
```

Summary

Concept	Description
Reactive Programming	Asynchronous, event-driven programming model
Reactive Streams	Defines Publisher, Subscriber, and Backpressure handling
Project Reactor	Java's popular reactive library (Mono, Flux)
Backpressure	Controls flow to prevent overload
Schedulers	Defines where work is executed (parallel(), boundedElastic())

Would you like more real-world examples or hands-on exercises? 🚀

Working with Java Modules (Java 9+)

Java introduced **modular programming** in **Java 9** with the **Java Platform Module System (JPMS)** to improve maintainability, scalability, and security. **Java 17** continues to support and enhance modularity, making it crucial for large-scale applications.

---

1. Introduction to Modular Programming

What is Modular Programming?

Modular programming is a software design approach where a system is divided into **separate, independent modules** that can be developed, tested, and maintained separately.

✅ **Encapsulation**: Hides implementation details

✅ **Reusability**: Modules can be used in different applications

✅ **Maintainability**: Easier debugging and updates

✅ **Scalability**: Large applications can be built efficiently

Why Java Modules? (JPMS - Java Platform Module System)

🔹 **Eliminates "JAR Hell" –** Prevents conflicts due to multiple versions of dependencies

🔹 **Improves Performance –** Java applications start faster by loading only required modules

🔹 **Enhances Security –** Restricts access to internal APIs

🔹 **Better Dependency Management –** Defines dependencies explicitly

---

2. Creating and Using Modules in Java

Key Components of Java Modules

Component	Description
module-info.java	Defines the module and its dependencies
exports	Makes a package available to other modules
requires	Specifies dependencies on other modules
opens	Allows reflection-based access (e.g., for frameworks like Spring)

Step 1: Creating a Simple Java Module

1️⃣ **Create a Directory Structure**

myapp/

├── moduleA/

```
│   ├── module-info.java
│   ├── com.example.modulea/
│   │   ├── Hello.java
```

2️⃣ Define module-info.java for Module A

```
module moduleA {
    exports com.example.modulea;  // Exposing this package
}
```

3️⃣ Create a Java Class in com.example.modulea

```
package com.example.modulea;

public class Hello {
    public static void sayHello() {
        System.out.println("Hello from Module A!");
    }
}
```

Step 2: Creating Another Module That Uses Module A

1️⃣ Directory Structure for Module B

```
myapp/
├── moduleA/
│   ├── module-info.java
│   ├── com.example.modulea/
```

```
│   │   ├── Hello.java
├── moduleB/
│   ├── module-info.java
│   ├── com.example.moduleb/
│   │   ├── MainApp.java
```

2️⃣ Define module-info.java for Module B

```
module moduleB {
    requires moduleA;  // Module B depends on Module A
}
```

3️⃣ Create MainApp.java in moduleB to Use Module A

```
package com.example.moduleb;

import com.example.modulea.Hello;

public class MainApp {
    public static void main(String[] args) {
        Hello.sayHello();  // Calls the method from Module A
    }
}
```

Step 3: Compiling and Running the Modules

📌 Compile Modules

javac -d out/moduleA moduleA/module-info.java moduleA/com/example/modulea/Hello.java

javac -d out/moduleB --module-path out -sourcepath moduleB moduleB/module-info.java moduleB/com/example/moduleb/MainApp.java

📌 **Run the Application**

java --module-path out -m moduleB/com.example.moduleb.MainApp

Output:

Hello from Module A!

---

3. Advanced Java Modularity in Java 17

Opening Packages for Reflection (opens)

Some frameworks (like Spring and Hibernate) require **reflection** to access private members.

To allow this, use the opens directive:

```
module moduleA {
    opens com.example.modulea;  // Allows reflection-based access
}
```

Using provides and uses for Service Providers

Java modules support **service providers** using provides and uses.

- 🔹 provides - Specifies an implementation
- 🔹 uses - Consumes the service

📌 **Example: Creating a Service Interface**

```
package com.example.service;
```

```
public interface GreetingService {
   void greet();
}
```

📌 **Module A Provides an Implementation**

```
module moduleA {
   exports com.example.service;
   provides com.example.service.GreetingService with
com.example.modulea.GreetingImpl;
}
```

📌 **Service Implementation in Module A**

```
package com.example.modulea;

import com.example.service.GreetingService;

public class GreetingImpl implements GreetingService {
   public void greet() {
      System.out.println("Hello from Service Provider!");
   }
}
```

📌 **Module B Uses the Service**

```
module moduleB {
   requires moduleA;
   uses com.example.service.GreetingService;
}
```

📌 Consuming the Service in Module B

```
package com.example.moduleb;


import com.example.service.GreetingService;

import java.util.ServiceLoader;


public class MainApp {

  public static void main(String[] args) {

    ServiceLoader<GreetingService>  loader =
ServiceLoader.load(GreetingService.class);

    loader.findFirst().ifPresent(GreetingService::greet);

  }

}
```

---

4. Summary

Feature	Description
Modules	Organizes code into self-contained units
module-info.java	Defines module dependencies and exports
exports	Allows other modules to use the package
requires	Declares dependencies on other modules
opens	Grants reflection-based access
provides & uses	Enables service providers

📌 Key Takeaways

✅ Java modules improve **security, maintainability, and dependency management**

✅ module-info.java is the core of Java's **modular system**

✅ exports and requires define **visibility and dependencies**

✅ provides and uses allow **service-based programming**

Would you like a **hands-on exercise** or more **real-world use cases**? 🚀

Java Networking

Networking in Java is handled using the **java.net** package, which provides classes for working with **TCP, UDP, and HTTP communication**. Java allows applications to communicate over networks using sockets, URLs, and protocols.

---

1. Basics of Networking in Java

What is Networking?

Networking enables **two or more devices** to communicate and share resources over the **internet or a local network**.

Key Networking Concepts

- **IP Address –** Unique identifier of a device on a network
- **Port Number –** Logical endpoint for network communication
- **Socket –** Endpoint for sending and receiving data
- **Protocol –** Rules defining how data is transmitted (e.g., TCP, UDP, HTTP)

Common Java Networking Classes (java.net package)

Class/Interface	Description
InetAddress	Represents an IP address
Socket	Client-side TCP communication
ServerSocket	Server-side TCP communication
DatagramSocket	UDP socket for sending/receiving data
URL	Handles HTTP communication
HttpURLConnection	Sends HTTP requests and receives responses

---

2. TCP and UDP Communication in Java

🔹 **TCP (Transmission Control Protocol)**

- **Connection-oriented** (establishes a connection before data transfer)
- **Reliable** (ensures data is delivered in order and without errors)
- **Slower but accurate**

Example: TCP Client-Server Communication

📌 **Step 1: Create a TCP Server**

```
import java.io.*;
import java.net.*;

public class TCPServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(5000);
        System.out.println("Server is waiting for a client...");

        Socket socket = serverSocket.accept(); // Accept connection
        System.out.println("Client connected!");

        BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

        String message = input.readLine(); // Receive message
        System.out.println("Client: " + message);

        output.println("Hello from Server!"); // Send response

        socket.close();
        serverSocket.close();
```

```
    }
}
```

📌 Step 2: Create a TCP Client

```
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 5000);

        PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        output.println("Hello Server!"); // Send message

        String response = input.readLine(); // Receive response
        System.out.println("Server: " + response);

        socket.close();
    }
}
```

- **Run the server first, then the client**
- The server waits for a client to connect and exchanges messages

---

- **UDP (User Datagram Protocol)**

- **Connectionless** (no prior connection setup)
- **Faster but less reliable**
- **Used for real-time applications (e.g., video streaming, online gaming)**

Example: UDP Client-Server Communication

📌 **Step 1: Create a UDP Server**

```
import java.net.*;

public class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(5000);
        byte[] buffer = new byte[1024];

        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
        serverSocket.receive(packet);  // Receive data

        String message = new String(packet.getData(), 0, packet.getLength());
        System.out.println("Client: " + message);

        String response = "Hello from UDP Server!";
        byte[] responseData = response.getBytes();

        DatagramPacket responsePacket = new DatagramPacket(responseData,
responseData.length, packet.getAddress(), packet.getPort());
        serverSocket.send(responsePacket);  // Send response

        serverSocket.close();
    }
}
```

📌 Step 2: Create a UDP Client

```
import java.net.*;


public class UDPClient {
   public static void main(String[] args) throws Exception {
      DatagramSocket clientSocket = new DatagramSocket();
      InetAddress serverAddress = InetAddress.getByName("localhost");


      String message = "Hello UDP Server!";
      byte[] buffer = message.getBytes();


      DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
serverAddress, 5000);
      clientSocket.send(packet);  // Send data


      byte[] responseBuffer = new byte[1024];
      DatagramPacket responsePacket = new DatagramPacket(responseBuffer,
responseBuffer.length);
      clientSocket.receive(responsePacket);  // Receive response


      String response = new String(responsePacket.getData(), 0,
responsePacket.getLength());
      System.out.println("Server: " + response);


      clientSocket.close();
   }
}
```

- **UDP is faster but may lose data**
- No explicit connection setup, just sending and receiving packets

---

3. HTTP Client in Java

- **Making HTTP Requests with HttpURLConnection**

Java provides HttpURLConnection for handling HTTP requests.

📌 **Example: Sending an HTTP GET Request**

```
import java.io.*;
import java.net.*;

public class SimpleHttpClient {
   public static void main(String[] args) throws IOException {
      URL url = new URL("<https://jsonplaceholder.typicode.com/todos/1>");
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();

      connection.setRequestMethod("GET");
      connection.setRequestProperty("Accept",  "application/json");

      int responseCode = connection.getResponseCode();
      if (responseCode == HttpURLConnection.HTTP_OK) {
         BufferedReader input = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
         String line;
         StringBuilder response = new StringBuilder();
         while ((line = input.readLine()) != null) {
            response.append(line);
         }
         input.close();
```

```
            System.out.println("Response: " + response);
        } else {
            System.out.println("GET request failed!");
        }

        connection.disconnect();
    }
}
```

🔹 Using Java 11+ HttpClient (Recommended)

Java 11 introduced a modern HttpClient API for **asynchronous and synchronous** HTTP requests.

📌 Example: Sending an HTTP GET Request with Java 11 HttpClient

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class ModernHttpClient {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("<https://jsonplaceholder.typicode.com/todos/1>"))
                .GET()
                .build();
```

```
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        System.out.println("Response: " + response.body());
    }
}
```

- 🔹 **Modern, Non-blocking, and Easy to Use**
- 🔹 Supports **synchronous** (send()) and **asynchronous** (sendAsync())

📌 **Summary**

Topic	Key Takeaways
Java Networking Basics	Uses java.net package for network communication
TCP (Socket Programming)	Reliable, connection-oriented, used for client-server apps
UDP (DatagramSocket)	Fast, connectionless, used for real-time applications
HTTP Communication	Java 11 HttpClient is modern and non-blocking

Would you like **more advanced networking examples**, such as **WebSockets or REST API integration**? 🚀

Reverse Engineering Concepts in Java

Reverse engineering is the process of analyzing a system to understand its design, functionality, and implementation without having access to its source code. In Java, reverse engineering is often used for **debugging, security analysis, understanding legacy code, and improving software.** However, it can also be used maliciously, so developers take measures like **code obfuscation** to protect their software.

---

1. Introduction to Reverse Engineering

Why is Reverse Engineering Important?

✅ **Understanding legacy code –** Helps developers maintain and update old systems

✅ **Debugging and security analysis** – Identifies vulnerabilities and exploits

✅ **Recovering lost source code –** Retrieves logic from compiled binaries

✅ **Software interoperability** – Helps integrate with undocumented APIs

✅ **Malware analysis** – Helps identify threats in applications

Common Reverse Engineering Techniques in Java

🔹 **Decompilation** – Converts Java bytecode back into source code

🔹 **Bytecode Analysis** – Examines .class files at the bytecode level

🔹 **Reflection & Introspection** – Dynamically inspects classes, methods, and fields at runtime

🔹 **Debugging & Code Analysis** – Traces execution and analyzes behavior

🔹 **Code Obfuscation & Deobfuscation** – Protects or reveals the true meaning of the code

---

2. Decompilation in Java

What is Decompilation?

Decompilation is the process of converting Java bytecode (.class files) back into readable Java source code.

Decompilation Tools

Tool	Description
JD-GUI	Lightweight GUI-based Java decompiler
FernFlower	IntelliJ's built-in decompiler
CFR	Powerful open-source Java decompiler
Procyon	Handles Java 8+ features well
JAD (deprecated)	Older decompiler, still used in some cases

Example: Decompiling a Java Class

1. **Compile a Simple Java Program**

```
public class Example {
  public static void main(String[] args) {
    System.out.println("Hello, Reverse Engineering!");
  }
```

}

javac Example.java

1. **Decompile the Generated Example.class**
    - Open Example.class in **JD-GUI** or **CFR** to view the source code.

---

3. Java Bytecode Analysis

What is Java Bytecode?

Java bytecode is an **intermediate representation** of Java programs, executed by the **Java Virtual Machine (JVM)**.

Analyzing Bytecode with javap

The javap tool disassembles .class files to reveal **bytecode instructions**.

javap -c Example.class

Example Output

```
public class Example {
  public static void main(java.lang.String[]);
    Code:
      0: getstatic     #2   // Field java/lang/System.out:Ljava/io/PrintStream;
      3: ldc           #3   // String Hello, Reverse Engineering!
      5: invokevirtual #4   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      8: return
}
```

- getstatic → Loads System.out
- ldc → Loads a constant ("Hello, Reverse Engineering!")
- invokevirtual → Calls println()

Bytecode Editing Tools

- **Javassist –** Dynamically modifies bytecode at runtime
- **ASM –** Framework for analyzing and modifying bytecode

---

4. Reflection and Introspection in Java

What is Reflection?

Reflection allows **dynamic inspection and modification** of Java classes, methods, and fields at runtime.

Example: Inspecting Methods of a Class

```
import java.lang.reflect.Method;


public class ReflectionExample {
   public static void main(String[] args) {
      Class<?> clazz = String.class;
      Method[] methods = clazz.getDeclaredMethods();
      for (Method method : methods) {
         System.out.println(method.getName());
      }
   }
}
```

Output:

```
charAt
compareTo
concat
equals
hashCode
indexOf
```

...

- **Useful for debugging and analyzing unknown classes**
- **Used in frameworks like Spring and Hibernate**

---

5. Debugging and Code Analysis Techniques

Debugging Tools

Tool	**Usage**
JDB (Java Debugger)	Command-line debugging
Eclipse/IntelliJ Debuggers	GUI-based debugging
VisualVM	Performance profiling
Ghidra	Security analysis

Example: Using JDB for Debugging

jdb Example

Commands:

- stop in Example.main → Sets a breakpoint
- run → Runs the program
- print var → Prints the value of a variable

---

6. Code Obfuscation and Deobfuscation

Why Obfuscate Java Code?

- Protects against **reverse engineering**
- Prevents **intellectual property theft**
- Increases security by **hiding logic**

Obfuscation Tools

Tool	Description
ProGuard	Popular Java bytecode obfuscator
Zelix KlassMaster	Advanced commercial obfuscator
Allatori	Used for commercial protection
DashO	High-security obfuscation

Example: Using ProGuard

1. Install ProGuard and create a config file:

```
proguard @proguard-rules.pro -injars Example.jar -outjars Example_obfuscated.jar
```

1. **Check Obfuscated Class Names**
    - Original: public class Example { }
    - Obfuscated: public class a { }

---

7. Working with Legacy Code and APIs

Challenges with Legacy Code

- Lack of documentation
- Old, unreadable code
- Deprecated APIs

Techniques for Understanding Legacy Code

✅ **Decompile classes** to view the source code

✅ **Use bytecode analysis** to inspect compiled binaries

✅ **Enable logging/debugging** to trace execution

✅ **Refactor gradually** to modernize code

---

8. Software Security through Reverse Engineering

How Reverse Engineering Helps Security?

- Identifies vulnerabilities in **Java applications**

- 🔹 Detects **malware and security exploits**
- 🔹 Analyzes **proprietary software for weaknesses**

Common Attacks on Java Applications

Attack	Description
Decompilation	Extracting Java source code from .class files
Bytecode Injection	Modifying bytecode to alter behavior
Reflection Exploits	Accessing private methods and fields
Dynamic Debugging	Analyzing execution flow to find weaknesses

---

📌 **Summary**

Concept	Key Takeaways
Decompilation	Converts Java .class files back into source code
Bytecode Analysis	Uses javap and ASM to analyze compiled Java
Reflection	Dynamically inspects and modifies Java classes
Debugging	Tools like JDB and VisualVM help analyze code behavior
Obfuscation	Protects Java applications from reverse engineering
Legacy Code	Techniques for analyzing and modernizing old code
Security	Helps detect vulnerabilities and prevent attacks

Would you like **hands-on exercises** or **real-world case studies** on reverse engineering? 🚀

Java Language-Specific Features in Java 17 and Java 21

Java 17 (LTS) and Java 21 (LTS) introduced several powerful language features that enhance **readability, performance, and maintainability**. Let's explore these features in detail.

Java 17 Features (LTS Release)

1. Sealed Classes and Interfaces

- **Restricts which classes can extend a class or implement an interface**
- Improves **security and maintainability**

Example: Sealed Class

```
sealed class Vehicle permits Car, Bike { }

final class Car extends Vehicle { }

final class Bike extends Vehicle { }
```

- The sealed keyword allows only Car and Bike to extend Vehicle.
- Other classes **cannot** extend Vehicle.

---

2. Pattern Matching for instanceof

- Simplifies type checking and casting
- Reduces **boilerplate code**

Example: Before Java 17

```
if (obj instanceof String) {
   String s = (String) obj;
   System.out.println(s.length());
}
```

Example: Java 17 (Improved)

```
if (obj instanceof String s) {
   System.out.println(s.length());
}
```

✅ **No explicit type casting required!**

---

3. Text Blocks (Introduced in Java 13, Finalized in Java 15)

- Makes multi-line **strings** more readable
- No need for escape characters (\\n, \\")

Example: Without Text Blocks

```
String json = "{\\n" +
        "  \\"name\\": \\"John\\",\\n"  +
        "  \\"age\\": 30\\n" +
        "}";
```

Example: With Text Blocks

```
String json = """
  {
    "name": "John",
    "age": 30
  }
  """;
```

✅ **Cleaner and easier to read**

---

4. Records (Finalized in Java 16, Improved in Java 17)

- **Immutable data classes** with **auto-generated** constructors, getters, and toString().
- Reduces **boilerplate code**

Example: Defining a Record

```
record Person(String name, int age) { }
```

✅ No need to write **constructors, getters, equals(), hashCode(), toString()**

Usage

```
Person p = new Person("Alice", 25);
System.out.println(p.name()); // Alice
```

---

5. Switch Expression Enhancements

- **Arrow (->) syntax** removes break statements
- Supports **return values**

Example: Before Java 17

```
switch (day) {
  case "MONDAY":
  case "FRIDAY":
    System.out.println("Workday");
    break;
  default:
    System.out.println("Relax");
}
```

Example: Java 17 (Improved)

```
String type = switch (day) {
  case "MONDAY", "FRIDAY" -> "Workday";
  default -> "Relax";
};
```

✅ **No break needed**

✅ **Returns a value directly**

---

6. Hidden Classes

- 🔹 Classes **not discoverable via reflection**
- 🔹 Used for **dynamic code generation** (e.g., proxies, runtime compilation)

Use Case

- Java frameworks like **Spring and Hibernate** use dynamically generated classes.
- Hidden classes **improve security** by **preventing unauthorized access**.

Java 21 Features (LTS Release)

1. String Templates (Preview Feature)

- 🔹 **Dynamic string formatting with placeholders**
- 🔹 Similar to **Python's f-strings**

Example: Before Java 21

```
String name = "Alice";
String greeting = "Hello, " + name + "!";
```

Example: Java 21 (Improved)

```
String name = "Alice";
String greeting = STR."Hello, \\{name}!";
```

✅ **More readable and avoids concatenation**

---

2. Sequenced Collections

- 🔹 Introduces **SequencedCollection, SequencedSet, and SequencedMap**
- 🔹 **Supports bidirectional access** (first/last elements)

Example: Working with SequencedCollection

```
SequencedCollection<String> list = new ArrayList<>();
list.add("A");
list.add("B");
```

```
list.add("C");


System.out.println(list.getFirst()); // A
System.out.println(list.getLast());  // C
```

✅ Useful for **linked lists, queues, and ordered maps**

---

3. Pattern Matching for switch and Record Patterns

- Enables **pattern matching inside switch statements**
- Simplifies **working with Records**

Example: Before Java 21

```
if (obj instanceof Person) {
   Person p = (Person) obj;
   System.out.println(p.name());
}
```

Example: Java 21 (Improved)

```
switch (obj) {
   case Person p -> System.out.println(p.name());
   case String s -> System.out.println("String: " + s);
   default -> System.out.println("Unknown");
}
```

✅ **No explicit casting needed**

✅ **More concise**

---

4. Virtual Threads (From Project Loom)

- 🔹 Lightweight **threads** managed by the JVM
- 🔹 Allows **millions of threads** without performance issues

Example: Creating Virtual Threads

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
   executor.submit(() -> System.out.println("Hello from virtual thread"));
}
```

✅ **Efficient for high-concurrency applications**

✅ **Improves performance for web servers and async processing**

📌 **Summary Table: Java 17 vs. Java 21 Features**

Feature	Java 17	Java 21
Sealed Classes	✅ Yes	❌ No New Changes
Pattern Matching (instanceof)	✅ Yes	✅ Extended for switch
Text Blocks	✅ Yes	❌ No New Changes
Records	✅ Yes	✅ Extended with Pattern Matching
Switch Expression Enhancements	✅ Yes	✅ Supports Pattern Matching
Hidden Classes	✅ Yes	❌ No New Changes
String Templates	❌ No	✅ New Feature
Sequenced Collections	❌ No	✅ New Feature
Virtual Threads	❌ No	✅ New Feature

🚀 **Why Upgrade to Java 21?**

✅ **Improved Readability** (String Templates, Pattern Matching)

✅ **Better Performance** (Virtual Threads)

✅ **Simplified Collections API** (Sequenced Collections)

✅ **More Expressive Code** (Records & Switch Enhancements)

Would you like **hands-on coding exercises** for these features? 😊

Introduction to Java Database Connectivity (JDBC)

📌 **What is JDBC?**

Java Database Connectivity (**JDBC**) is an **API** (Application Programming Interface) that allows Java applications to interact with **relational databases** such as **MySQL, PostgreSQL, Oracle, and SQL Server**.

🔹 JDBC acts as a **bridge** between Java applications and databases.

🔹 Provides methods for **executing SQL queries, handling transactions, and managing connections**.

1️⃣ **JDBC Architecture**

JDBC follows a **4-layered architecture**:

Layer	Description
JDBC API	Provides methods to interact with the database (e.g., Connection, Statement, ResultSet)
JDBC Driver Manager	Manages database drivers and establishes connections
JDBC Driver	Communicates with the database
Database	The actual database that stores data

2️⃣ **Steps to Use JDBC**

To connect Java with a database using JDBC, follow these steps:

Step 1: Load the JDBC Driver

Before Java 8:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- **Since Java 8, explicitly loading the driver is not needed** (JDBC automatically loads it).

Step 2: Establish a Connection

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "username", "password");
```

- Replace "mydatabase", "username", and "password" with actual values.
- The getConnection() method establishes a connection to the database.

Step 3: Create a Statement

```
Statement stmt = conn.createStatement();
```

- The Statement object is used to execute SQL queries.

Step 4: Execute an SQL Query

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

- The executeQuery() method runs **SELECT** queries and returns a ResultSet.

Step 5: Process the Results

```
while (rs.next()) {
   System.out.println("User ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}
```

- rs.next() moves to the next row.
- Retrieve data using column names (getInt(), getString(), etc.).

Step 6: Close the Connection

```
rs.close();
stmt.close();
conn.close();
```

- Closing connections **prevents memory leaks**.

3️⃣ Executing Different SQL Queries

1. SELECT Query

```
String query = "SELECT * FROM employees WHERE department = 'IT'";

ResultSet rs = stmt.executeQuery(query);
```

2. INSERT Query

```
String insertQuery = "INSERT INTO employees (name, age, department) VALUES ('John Doe', 30, 'HR')";

stmt.executeUpdate(insertQuery);
```

- executeUpdate() is used for **INSERT, UPDATE, DELETE** queries.

3. UPDATE Query

```
String updateQuery = "UPDATE employees SET age = 35 WHERE name = 'John Doe'";

stmt.executeUpdate(updateQuery);
```

4. DELETE Query

```
String deleteQuery = "DELETE FROM employees WHERE name = 'John Doe'";

stmt.executeUpdate(deleteQuery);
```

4️⃣ Handling Transactions in JDBC

- **By default, JDBC runs in auto-commit mode** (each SQL statement is committed immediately).
- You can **disable auto-commit** to **group multiple queries into a single transaction**.

Example: Handling Transactions

```
try {
    conn.setAutoCommit(false); // Start transaction

    stmt.executeUpdate("INSERT INTO accounts (name, balance) VALUES ('Alice', 5000)");
    stmt.executeUpdate("INSERT INTO accounts (name, balance) VALUES ('Bob', 3000)");

    conn.commit(); // Commit transaction
} catch (SQLException e) {
    conn.rollback(); // Rollback changes if an error occurs
    e.printStackTrace();
} finally {
    conn.setAutoCommit(true); // Restore auto-commit mode
}
```

✅ **Ensures that both queries execute successfully before committing.**

✅ **If an error occurs, rollback prevents partial data updates.**

5️⃣ Using PreparedStatement for Security

🔹 PreparedStatement is used to **prevent SQL injection attacks** and improve performance.

Example: Using PreparedStatement

```
String query = "INSERT INTO users (name, email) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setString(1, "Alice");
pstmt.setString(2, "alice@example.com");
pstmt.executeUpdate();
```

✅ Prevents **SQL injection**

✅ **Faster execution** due to precompiled SQL

6️⃣ Using Connection Pooling (Best Practice for Large Applications)

🔹 **Connection pooling** improves performance by **reusing database connections** instead of creating a new one every time.

🔹 **Apache DBCP, HikariCP, and C3P0** are commonly used connection pools.

Example: Using HikariCP

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase");
config.setUsername("root");
config.setPassword("password");
HikariDataSource ds = new HikariDataSource(config);

Connection conn = ds.getConnection();
```

✅ **Improves performance**

✅ **Efficient resource utilization**

📌 Summary of Key JDBC Concepts

Concept	Description
JDBC API	Connects Java applications to relational databases
JDBC Driver	Acts as a bridge between Java and databases
Statement	Executes SQL queries (executeQuery, executeUpdate)
PreparedStatement	Prevents SQL Injection and improves performance
Transactions (commit/rollback)	Ensures atomicity and consistency

Concept	Description
Connection Pooling	Improves efficiency by reusing connections

---

🚀 **Why Learn JDBC?**

✅ Essential for **database-driven applications**

✅ Used in frameworks like **Spring JDBC, Hibernate**

✅ Critical for **enterprise applications, banking, and e-commerce systems**

Would you like a **hands-on project** using JDBC, such as a **CRUD (Create, Read, Update, Delete) application**? 😊